Cairo University
Faculty of Engineering
Department of Computer Engineering

Inferex

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by
Mostafa Elsayed

Supervised by
Dr. Sandra Wahid

July 2025

Abstract

Inferex is an intelligent desktop application designed to help users recall and retrieve information they have previously seen on their computers, whether from a video, document, or website, without the need to manually save or organize it. In today's digital world, users interact with large volumes of information every day, yet remembering specific details from past activities remains a constant challenge. Most AI assistants currently available require users to manually provide the context of the question to receive an accurate and personalized answer, which limits their convenience and effectiveness. Inferex solves this problem by silently and privately building a local context from the user's screen activity, enabling it not only to retrieve previously seen content but also to answer related questions intelligently without requiring additional input from the user. The system operates entirely offline, ensuring complete privacy by keeping all data on the device. It captures screenshots periodically, segments them into meaningful parts, and stores them in a vector database for semantic retrieval. Lightweight algorithms and compressed models are used to guarantee smooth performance even on personal devices with limited hardware resources. Inferex offers a private, context-aware, and searchable memory of the user's digital interactions, making it a powerful tool for productivity, learning, and everyday information recall.

الملخص

إنفريكس (Inferex) هو تطبيق ذكي لسطح المكتب صُمم لمساعدة المستخدمين على تذكّر واسترجاع المعلومات التي شاهدوها سابقًا على أجهزتهم، سواء كانت من فيديو، مستند، أو موقع إلكتروني، دون الحاجة إلى حفظها أو تنظيمها يدويًا. في عالمنا الرقمي اليوم، يتفاعل المستخدمون يوميًا مع كميات ضخمة من المعلومات، ومع ذلك فإن تذكّر التفاصيل الدقيقة من الأنشطة السابقة لا يزال تحديًا مستمرًا. تعتمد معظم أدوات الذكاء الاصطناعي المتاحة حاليًا على أن يقوم المستخدم بتوفير سياق السؤال يدويًا للحصول على إجابة دقيقة وشخصية، مما يقلل من كفاءتها ويجعل استخدامها أقل سهولة. يحلّ إنفريكس هذه المشكلة من خلال إنشاء سياق محلي تلقائي وخاص بناءً على نشاط الشاشة لدى المستخدم، مما يمكّنه ليس فقط من استرجاع المحتوى الذي تم عرضه سابقًا، بل أيضًا من الإجابة على الأسئلة المتعلقة به بذكاء ودون الحاجة إلى إدخال أي معلومات إضافية. يعمل النظام بشكل كامل دون اتصال بالإنترنت، مما يضمن خصوصية المستخدم من خلال إبقاء جميع البيانات محفوظة على الجهاز فقط. يلتقط التطبيق لقطات شاشة بشكل دوري، ويقوم بتقسيمها إلى أجزاء ذات معنى، ثم يخزنها في قاعدة بيانات يمكن البحث فيها بطريقة دلالية. كما تم استخدام خوارزميات خفيفة ونماذج مضغوطة لضمان أداء سلس حتى على الأجهزة الشخصية ذات الموارد المحدودة. يقدم إنفريكس ذاكرة رقمية خاصة، مدركة للسياق، وقابلة للبحث، مما يجعله أداة قوية لزيادة الإنتاجية، وتسهيل التعلّم، واسترجاع المعلومات في الحياة اليومية.

ACKNOWLEDGMENT

I thank Dr. Sandra Wahid for her clear guidance, valuable feedback on each design iteration, and timely support whenever challenges arose. Her consistent encouragement and expertise were essential to the success of this project.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

BERT	Bidirectional Encoder Representations from Transformers
LLM	Large Language Model
Q&A	Question and Answering
QA	Question Answering
SQuAD	Stanford Question Answering Dataset

Contacts

Team Members

Name	Email	Phone Number
Mostafa Elsayed Mohamed	mostafa.elsayed.2002@gmail.com	+2 01111278802

Supervisor

Name	Email	Number
Sandra Wahid	sandrawahid@hotmail.com	-

This page is left intentionally empty

Chapter 1: Introduction

In Inferex, my contributions spanned three main modules: input processing, extractive question answering, and LLM compression. In the input processing module, I worked on converting paragraphs generated by the screen analyzer into vector embeddings, which were then stored in the backend to enable efficient retrieval. For the extractive question answering module, I integrated a custom loss function into the Basic BERT model to improve its handling of unanswerable questions, thereby enhancing its robustness and accuracy. In the LLM compression module, I analyzed the model weights to assess the feasibility of applying our chosen compression technique and implemented a GPU kernel capable of decompressing weights on the fly, which enhances inference time by reducing memory usage and maintaining high performance.

1.1. Document Organization

This section outlines the structure of the report and provides a brief overview of the upcoming chapters. Chapter 2 presents a detailed explanation of my contributions to the modules I worked on. Chapter 3 discusses the methods used to test and evaluate these contributions. Chapter 4 offers a brief conclusion, summarizing the experience I gained and the challenges I encountered throughout the project.

Chapter 2: System Design and Architecture

In this chapter, I provide a detailed explanation of my contributions to the project.

2.1. Input Processing Module: Handling Text

2.1.1. Functional Description

The input Processing Module serves as the primary data ingestion and preparation pipeline for the system. Its core function is to receive raw screen content, captured and segmented into a structured JSON format by the upstream Screen Analyzer module.

My contribution focuses on processing the input text. I began by removing sensitive information, followed by segmenting the content into manageable sentences to optimize it for semantic search. These refined sentences were then converted into numerical vector representations (embeddings). The resulting embeddings were stored in a vector database, enabling contextually-aware retrieval—a critical component for the effectiveness of the question answering system.

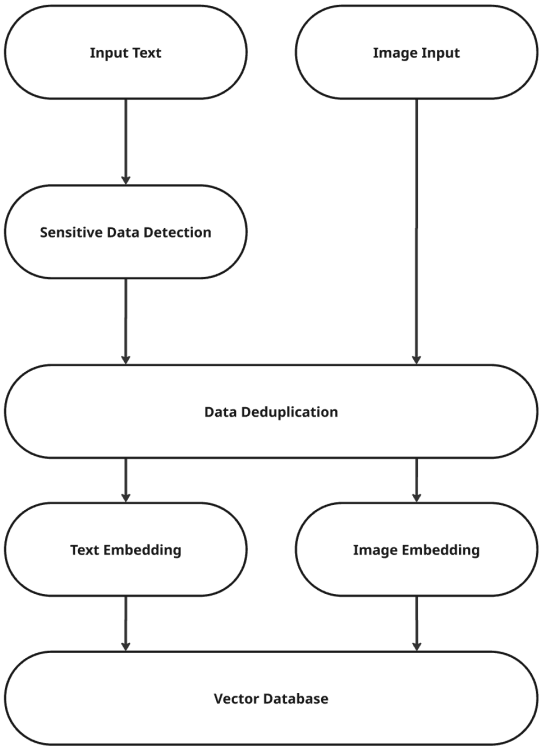

Figure 2.1: Input processing block diagram

2.1.2. Modular Decomposition

The Input Processing Module is broken down into several fine-grained sub-modules, each handling a specific stage of the data pipeline.

1. **Sensitive Data Detection:** This is the first processing step for any text. To ensure user privacy and data protection, this sub-module employs a rule-based detection system. It scans incoming text for patterns matching common sensitive information, such as social security numbers, dates of birth, phone numbers, email addresses, and credit card numbers. Any detected sensitive data is immediately masked by replacing it with the placeholder `<Sensitive Data>` before it is passed to any subsequent component or stored.

2. **Data Deduplication:** To maintain an efficient and clean database, this component prevents the storage of redundant information. Since screen captures can be repetitive, this sub-module checks if a piece of content already exists. It splits the input text into individual sentences, converts each sentence into an embedding, and queries the vector database for highly similar vectors that share the same source metadata (e.g., URL or file name). If a near-duplicate is found, the new sentence is discarded, saving storage space and reducing noise during retrieval.

3. **Sentence Embedding:** This sub-module is responsible for converting textual sentences into meaningful vector representations. It uses the `paraphrase-MiniLM-L6-v2` sentence-transformer model. The process is as follows:

    - **Tokenization:** The input sentence is broken down into words or sub-words (tokens).
    - **Embedding Generation:** Tokens are mapped to IDs and converted into input vectors, which are combined with positional embeddings to preserve word order.
    - **Transformer Layers:** The data flows through the model's six transformer layers, where a self-attention mechanism weighs the importance of different tokens in relation to each other, capturing deep contextual meaning.

2.2. Extractive Question Answering Module: Fine-Tuning using a Custom Loss Function

2.2.1. Functional Description

It is a natural language processing task where a model finds the answer to a question directly within a given text. Instead of generating a new answer, the model identifies and extracts the specific span of text —a word or a sequence of words— that contains the correct information.

For example:
context: "The Amazon rainforest is the largest tropical rainforest in the world, covering an area of approximately 6.7 million square kilometers. It is home to an estimated 390 billion individual trees divided into 16,000 species."

Question: "How many individual trees are in the Amazon rainforest?"
Answer: "390 billion”

2.2.2. Modular Decomposition

The Extractive Question Answering module is designed around a fine-tuned BERT (Bidirectional Encoder Representations from Transformers) model. The architecture can be broken down into the following core components:

1. BERT Base Model: The foundation of the module is a pre-trained BERT model. BERT's bidirectional nature allows it to read an entire sequence of text at once, providing a deep contextual understanding of the relationship between the question and the context, which is critical for identifying the correct answer span.

2. Fine-Tuning on SQuAD v2: The base model is fine-tuned on the Stanford Question Answering Dataset (SQuAD) v2. This dataset is specifically chosen because it contains both questions that have an answer within the provided context and questions that are unanswerable, which is essential for building a robust real-world system.

3. Custom Loss Function: To improve the model's ability to identify unanswerable questions, a custom loss function was implemented. This function is a weighted combination of two separate loss components, replacing the standard cross-entropy loss used in the default BERT for question answering.

a. Modified QA Loss (LQA): This is an adaptation of the standard cross-entropy loss. For answerable questions, it functions normally. However, for unanswerable questions, instead of forcing the model to predict the [CLS] token, the ground truth label for all tokens is set to a uniform probability of $\frac{1}{n}$(where n is the sequence length). This prevents the model from incorrectly favoring any single token when no answer exists. The loss is calculated as:

$$L_{QA} = -\sum_{k=1}^{n} y_{sk} \log\left(\frac{\exp(s_k)}{\sum_{i=1}^{n}\exp(s_i)}\right) - \sum_{k=1}^{n} y_{ek} \log\left(\frac{\exp(e_k)}{\sum_{i=1}^{n}\exp(e_i)}\right)$$

Where:

Here, s_k and e_k are the model's output scores for the k-th token being the start and end of the answer, while ys_k and ye_k are the ground truth labels.

Figure 2.2: Question Answering Loss

b. Sequence Tagging Loss (L_{Tag}): This component treats the task as a binary classification for each token. It encourages the model to output negative scores for all possible start and end tokens when a question is unanswerable (all labels are 0). For answerable questions, the correct start and end tokens are labeled 1. This explicitly teaches the model a mechanism to signal that no valid answer span can be found.

$$L_{\text{Tag}} = -\sum_{k=1}^{n} \left(y_k^s \log\sigma(s_k) + (1-y_k^s)\log(1-\sigma(s_k))\right) - \sum_{k=1}^{n} \left(y_k^e \log\sigma(e_k) + (1-y_k^e)\log(1-\sigma(e_k))\right)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Figure 2.3: Sequence Tagging Loss

c. Overall Loss: The final loss is a weighted sum of these two components, allowing for fine-tuning of their relative importance. The final function is:

$$L_{total} = \lambda_{QA} L_{QA} + \lambda_{Tag} L_{Tag}$$

Where:

the hyperparameters were set to $\lambda_{QA} = 2$ and $\lambda_{Tag} = 1$.

Figure 2.4: Overall Loss

The integration of the new loss function led to an improvement in the F1 score for unanswerable questions, while the overall F1 score remained nearly unchanged.

Table 2.1: Extractive question answering with the new loss function

	Has Answer	**No Answer**	**Total**
F1 Score	74.17%	74.93%	74.55%

Table 2.2: Extractive question answering with the standard cross-entropy loss

	Has Answer	**No Answer**	**Total**
F1 Score	78.9%	69.9%	74.4%

2.3. LLM Compression Module: Analyzing LLM Weights and Implementing a Custom GPU Kernel

2.3.1. Functional Description

The purpose of this module is to reduce the size of the large language model (LLM), making it more suitable for deployment on user devices with limited computational resources. By compressing the model, this module enables faster loading, lower memory usage, and efficient inference on a wider range of hardware, including personal computers.

My contribution focuses on analyzing the LLM's weights and implementing a custom kernel that decompresses these weights on the fly during inference.

2.3.2. Modular Decomposition

We applied lossless compression to the Qwen3-4B language model. The process began with an analysis of the model's weights using principles from information theory. The weights in Qwen3-4B are stored using the bfloat16 (brain floating point 16) format, a 16-bit floating-point representation commonly used in modern deep learning models. Unlike IEEE float16, bfloat16 maintains the same 8-bit exponent as float32 but reduces the mantissa to 7 bits, along with a 1-bit sign, enabling a wide dynamic range with reduced precision.

2.3.2.1. Analyzing LLM Weights

To identify compression opportunities, we applied Shannon's entropy rule to each component of the bfloat16 format, which quantifies the minimum number of bits needed, on average, to encode values from a given distribution.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

- $H(X)$ is the **entropy** of the random variable X
- x_i represents each possible value or symbol
- $p(x_i)$ is the probability of occurrence of x_i

Figure 2.5: Equation of Shannon entropy

The results of the weight analysis are summarized as follows:

Table 2.3: Difference between the minimum number of bits that are required to represent each component of a bfloat16 in Qwen3-4B and the actual representation.

	Sign	**Exponent**	**mantissa**
Bfloat16	1	8	7
Minimum number of bits	1	2.6533	6.9725

Based on this analysis, the exponent field, which originally uses 8 bits, can be effectively represented using just 3 bits without significant information loss. The sign and mantissa remain unchanged. As a result, the total number of bits required to represent each weight can be reduced from 16 bits to 11 bits, achieving a more compact and efficient encoding.

To aid in understanding, the following histograms illustrate the distribution of each component.

Sign Bit Histogram:

The histogram below illustrates the distribution of the sign bit. As shown, the sign bit is evenly distributed between 0 and 1, indicating that both positive and negative values are used frequently and consistently throughout the model weights. This confirms that the sign bit utilizes its full 1-bit range effectively, and thus, cannot be further compressed without loss.

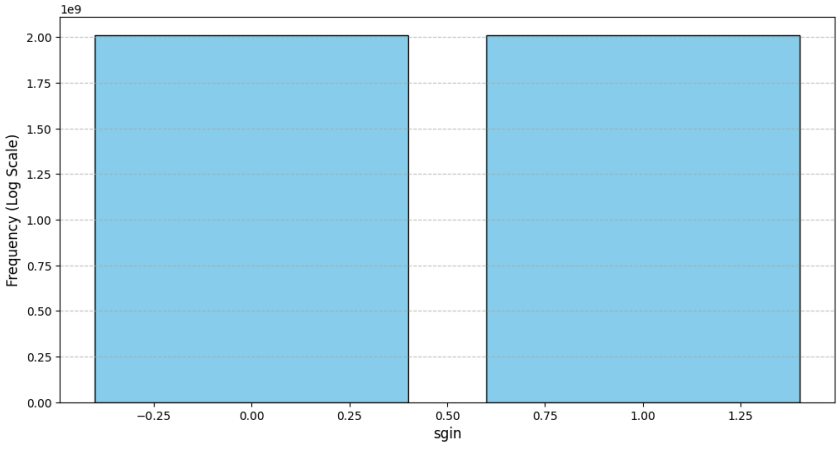

Figure 2.6: Distribution of the sign bit in Qwen3-4B

Mantissa Bits Histogram:

The histogram below shows the distribution of the mantissa bits. As observed, the values are spread across the full range of the 7-bit representation (0 to 127), indicating broad utilization of the mantissa field. This suggests that the mantissa carries significant information and is less compressible without loss.

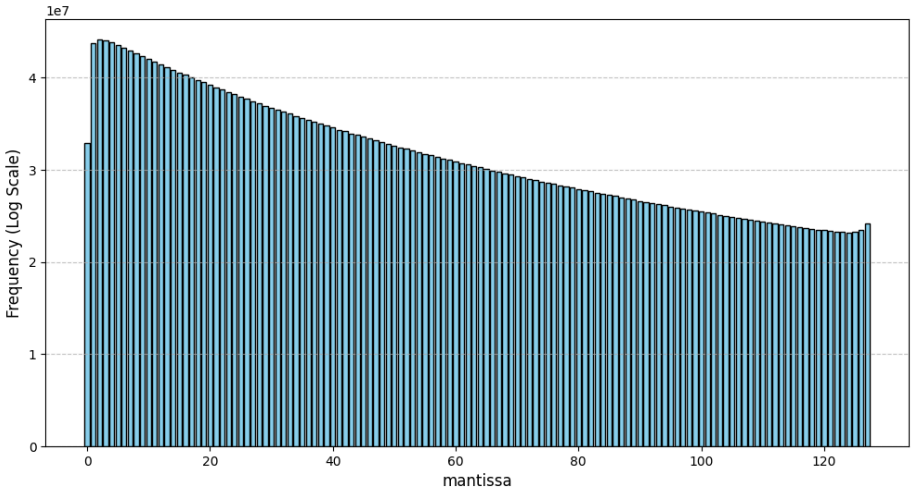

Figure 2.7: Distribution of Mantissa Bit in Qwen-4B)

Exponent Bits Histogram:

In contrast to the sign and mantissa, the histogram of the exponent bits reveals a narrow distribution, indicating that the exponent values do not fully utilize the available 8-bit range (0 to 255). This limited usage suggests significant redundancy and makes the exponent field a good candidate for lossless compression.

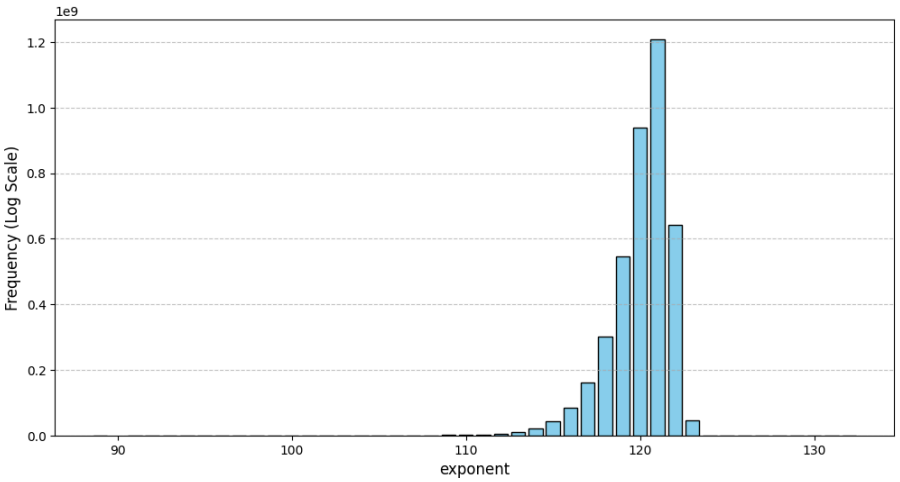

Figure 2.8: Distribution of Exponent Bit in Qwen3-4B

2.3.2.1. Implementing a Custom GPU Kernel

After encoding the exponent using Huffman coding, we needed to decode the weights during inference. If the decoding process were to be handled by the CPU, it would introduce a significant bottleneck: each time a feedforward pass occurs, the weights would need to be decompressed on the CPU, then transferred to the GPU for matrix multiplication, resulting in considerable overhead and latency.

To address this issue, we implemented the decompression directly on the GPU, just before the computation. This approach eliminates the need for weight transfer between the CPU and GPU and ensures faster inference.

The overall process is as follows:
When a feedforward pass is triggered, the compressed weights for the corresponding layer are passed to a custom GPU kernel, which performs on-the-fly decompression. Immediately after decompression, the matrix multiplication is executed to compute the output. Once the operation is complete, the temporarily decompressed weights are discarded, and the same process is repeated for each subsequent layer.

In parallel Huffman decoding, each thread is assigned 8 bytes of the compressed bitstream to process. The output of the compression includes the following components:

1. **Encoded Exponents**: The Huffman-compressed exponents.

2. **Gaps**: An array where each element holds a 5-bit value that indicates the offset each thread should apply before starting to read its assigned 8-byte segment.

3. **Output Positions**: An array specifying the starting output index for each thread's block.

4. **Sign and Mantissa**: A compact 8-bit representation that combines the sign (1 bit) and mantissa (7 bits).

5. **Lookup Tables (LUTs)**: Four LUTs are used to decode the compressed exponents efficiently.

6. **Code Lengths**: An array indicating the length of each Huffman-encoded exponent to guide accurate decoding.

The decoding process:

1. Read the first byte from the bitstream and use it to index into LUT1.
2. If the returned value is a reserved marker (indicating the code spans more than one byte), read the second byte and use it to index into LUT2.
3. This continues if necessary through LUT3 and LUT4, reading one additional byte at each step until a non-reserved value is returned.
4. Once the decoded exponent is obtained, the decoder accesses the code lengths LUT to determine the actual length of the Huffman code.
5. The bitstream pointer is then advanced by that length, and the next symbol is decoded in the same way.

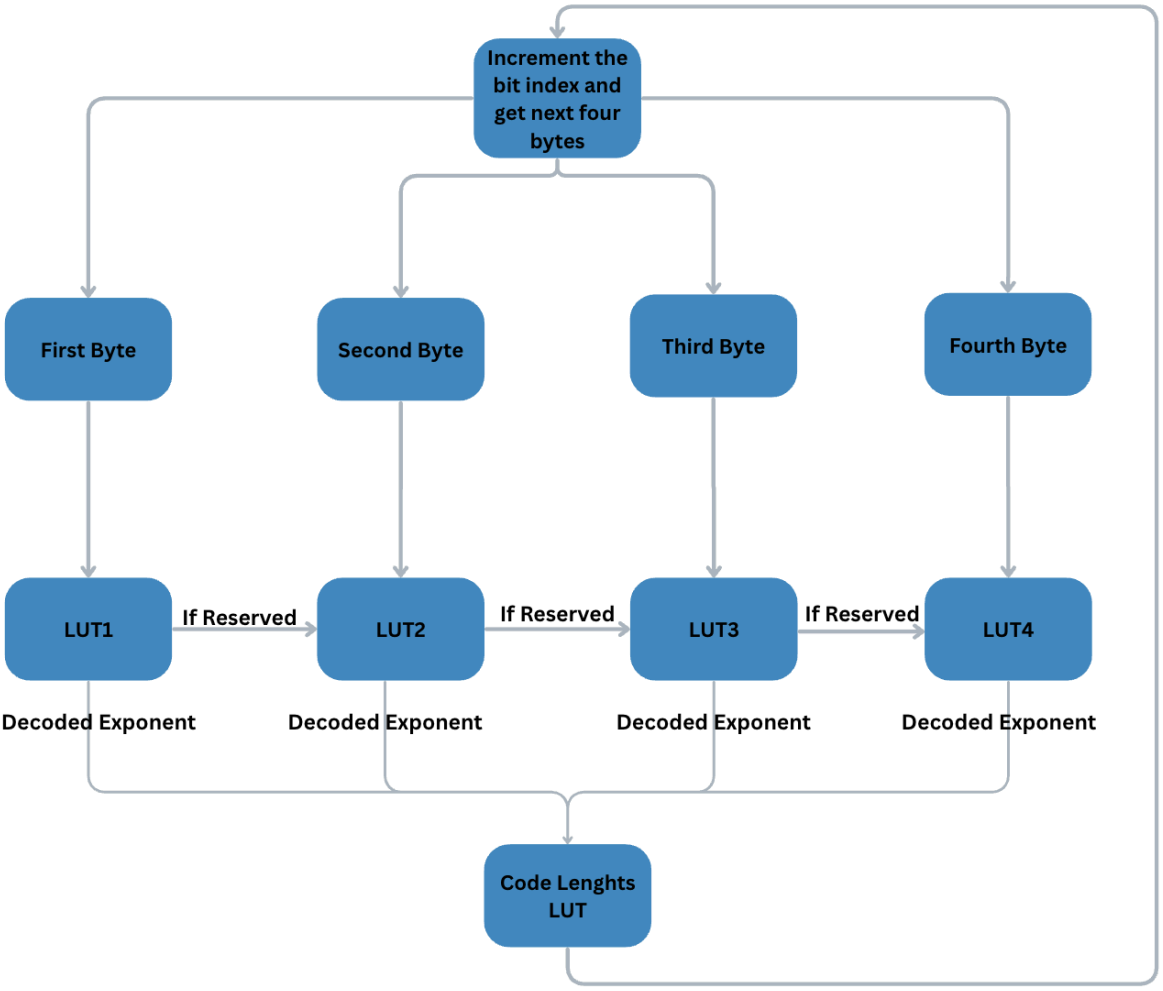

Figure 2.9: Decoding process using LUTs

Since the output positions array contains one entry per block, additional logic is required in the kernel to allow each thread to determine its correct starting position in the output.

To achieve this, the kernel is divided into two phases:

1. **Phase One – Position Calculation**:
   Each thread counts the number of exponents it will decode from its assigned data segment. An exclusive prefix sum is then performed within the block, allowing each thread to compute its relative write offset. By adding the block's starting position to each thread's prefix sum result, we obtain the absolute output position for each thread.

2. **Phase Two – Decoding and Writing**:
   In this phase, the threads perform the actual decoding of their assigned data segments. Since each thread now knows its exact output position, it can efficiently write its results to the correct location without any conflicts.

Kernel Input:

encoded_exponent: an array of 8-bit values representing the Huffman-encoded exponents

mantissa_sign: an array of 8-bit values, where the mantissa and sign bits are concatenated

gaps: an array of 8-bit values (each using only 5 bits to indicate the decoding offset per thread)

output_block_positions: an array of 32-bit values representing the starting output position for each block

LUT1–LUT4: four prefix-free lookup tables used for Huffman decoding

code_lengths_LUT: a table indicating the bit length of each Huffman-encoded exponent

The kernel executes the following steps:

Phase 1 – Preparation and Position Calculation

1. **Load LUTs into shared memory**: All four LUTs and the code lengths LUT are loaded into **shared memory** to allow fast access during decoding.
2. **Determine decoding start offset**: Each thread reads its corresponding **5-bit offset** from the gaps array and uses it to determine the correct **starting bit position** within its assigned **8-byte segment** of the encoded exponent array.
3. **Initialize a shared count array**: A shared memory array (size = block size) is allocated to **track the number of exponents decoded** by each thread.
4. **Begin decoding loop**:
    - Each thread reads up to 4 bytes from its data segment.
    - It attempts to decode the exponent by indexing LUT1 with the first byte. If a reserved value is returned, the thread continues to LUT2, LUT3, or LUT4 using additional bytes until a valid decoded value is found.
    - The decoded exponent is then used to look up its length in code_lengths_LUT.
    - The bitstream pointer is incremented by this length.
    - The thread increments its count in the shared array, indicating it has decoded another exponent.
    - This loop continues until all exponents that start within the 8-byte segment have been decoded.
5. **Perform exclusive prefix sum**:
    - An exclusive scan (prefix sum) is performed on the shared count array to compute the relative output offset for each thread.
6. **Adjust with block position**:
    - The starting position of the block, from output_block_positions, is added to each thread's offset. Each thread now knows exactly where to write its decoded output.

Phase 2 – Final Decoding and Writing

7. **Repeat the decoding process**:
    - Each thread decodes its assigned exponents again, using the same logic as in Phase 1.

8. **Construct a full 16-bit value**:
    - For each decoded exponent, the thread concatenates the exponent with its corresponding mantissa and sign bits to reconstruct the full bfloat16 value.

9. **Write output**:
    - The reconstructed value is written to the output array at the position calculated in Phase 1.

Chapter 3: System Testing and Verification

In this chapter, I will demonstrate how I tested and verified the correctness and effectiveness of my contribution.

3.1. Testing Setup

The testing was conducted on a laptop with the following specifications:

- CPU: AMD Ryzen 7 5800H
- RAM: 16 GB
- GPU: NVIDIA RTX 3060
- OS: Windows 11
- Python version: 3.10

3.2. Testing Plan and Strategy

For the AI-based modules, the strategy involves splitting the data into training and validation sets, monitoring the loss on both, and selecting the model version that achieves low validation loss without signs of overfitting.

For the GPU kernel, the approach is to compare the original weights with the decompressed weights to ensure the compression and decompression processes preserve accuracy.

3.2.1. Module Testing

3.2.1.1. Text Processing:

After generating the embeddings, I performed manual testing by inserting the embedding of a specific text into the vector database, then attempting to retrieve it using semantically similar paragraphs to evaluate the effectiveness of the semantic search.

3.2.1.2. Extractive Question Answering:

The dataset was divided into training and validation sets. During training, I monitored the loss on both sets and selected the model version that achieved the lowest validation loss while avoiding overfitting.

3.2.1.3. LLM Compression:

To test the kernel, I applied it to decompress weights and then subtracted the resulting decompressed weights from the original model weights. I verified that the difference was zero, ensuring the decompression was accurate and lossless.

3.3. Testing Schedule

Testing was conducted immediately after the completion of each component to ensure its functionality and correctness before proceeding to the next stage.

Chapter 4: Conclusions and Faced Challenges

4.1. Faced Challenges

During the development of my part in the extractive question answering module, I faced challenges with the model's accuracy on unanswerable questions due to an imbalanced dataset. To resolve this, I fine-tuned the model using the SQuAD2 dataset, which includes more unanswerable examples, and modified the loss function to better prioritize them. This slightly reduced accuracy on answerable questions but greatly improved performance in detecting unanswerable ones.

4.2. Gained Experience

Throughout the project, I developed strong expertise in natural language processing and the integration of AI into intelligent interfaces. A key area of focus was extractive question answering (QA) using pre-trained models like BERT, which enabled the system to accurately identify and extract answers from a given context, similar to the methods employed in modern search engines such as Google. This work provided valuable hands-on experience with transfer learning and fine-tuning deep learning models.

In addition, I designed, implemented, and optimized custom loss functions, which offered deeper insight into the dynamics of model training and performance enhancement.

I also worked on compressing large language models (LLMs) to improve inference speed by implementing custom CUDA kernels. This effort helped me build practical skills in CUDA programming while expanding my understanding of LLM architectures and their optimization for deployment.

4.3. Conclusions

In this project, I enhanced the performance of an extractive question answering system by improving BERT’s accuracy on unanswerable questions. This was achieved through fine-tuning with the SQuAD2 dataset and customizing the loss function to better handle such cases. Additionally, I implemented a custom CUDA kernel for decompressing model weights on the fly, ensuring efficient inference performance without compromising accuracy. These contributions demonstrate the feasibility of deploying advanced language models on devices with limited resources while maintaining both speed and reliability.

References

[1] S. Q. Tran and M. Kretchmar, “Towards Robust Extractive Question Answering Models: Rethinking the Training Methodology,” *arXiv preprint arXiv:2409.19766*, 2024.

[2] T. Zhang, Y. Sui, S. Zhong, V. Chaudhary, X. Hu, and A. Shrivastava, “70% Size, 100% Accuracy: Lossless LLM Compression for Efficient GPU Inference via Dynamic-Length Float,” arXiv preprint arXiv:2504.11651, 2025.